Project 2:

Title:
Uno Card Game V.2

Class:
CSC 17A - 43396

Due Date:
June 6th, 2021

Author:
Aamir Khan

Introduction and Gameplay:

Title: Uno Card Game V.2

Objective:

Each player begins with a certain number of cards that make up their deck. The goal of the game is to use up all of your cards before your opponent does so. To use these cards, players are expected to match their cards with the card currently on the pile. By pulling special cards, players may sabotage one another in hopes of getting to the end of their cards first. Uno isn't a complicated game and can be played with people of all ages. It's a fun, family game that may be played with stakes on the line or for pure fun and family time. Let's now have a look at the rules.

The "How-To":

Cards can be matched in a variety of ways, and that is what we'll be going over shortly. A random card will be placed in the card pile between two players. Players take turns matching whatever card is currently on the card pile. Cards can be matched either by colors, values, or using special cards such as the Wild card, the Draw 1 card, or the Draw 2 card. To match by color, the cards need to have matching colors only. To match by value, the cards just need to have the same face value. As for using special cards, a Wild card may be used at any time throughout the game. This card can be used to set a new color to match, chosen by the one who used the card. A Draw 1 card forces the opponent to draw 1 card and add it to their deck of cards. A Draw 2 card forces the opponent to draw 2 cards. Draw 1 and 2 cards may only be used when the colors match or when the card that needs to be matched is also a Draw 1 or 2 card. For example, if we have a green Draw 1 and the other card is a blue Draw 1, these cards can be matched because they're both Draw 1's.

Overview:

For Project 2, I took Uno and added some advanced concepts into the mix. I included everything between Chapters 13-16 from Gaddis, such as Classes, Inheritance, Polymorphic Behavior, and so much more. The game works just as my Project 1 did, except this version demonstrates my knowledge of Object-Oriented Programming. The line count, with all the files and functions adding up, draws somewhere above the 1,000 line minimum count. Some other concepts that I included are: Constructors, Destructors, Friends, Static Members, UML, Abstract Classes, and whatever else was on the Project 2 Check-off Sheet. With that, this concludes my second project.

Stages of Development:

Version 1: Card Generator

In this version, I created a card generator that took numbers and set card properties using switch statements with these numbers. For my card properties, I used numbers to set colors, values, and descriptions for cards that were special, such as wild cards or draw cards. The purpose of this version was as simple as it sounds: to get the basics up and running with respect to Uno cards.

Version 2: UnoCard Class

In this version, I threw in whatever I had into an UnoCard Class. A single UnoCard object would represent a single Uno card with a color, value, and description if necessary. I had my members declared private and several public member functions that would allow me to work with an object's data. The most important function was the set card function that would generate a random card and store its properties in an UnoCard class object. I had a simple display function that took data and displayed it in a format that was clear. I took advantage of getter functions to display such data. I used a setter function to set the color of a card and used this function to set a new color to match if we ever used a wild card throughout the program.

Version 3: Deck Double Pointer

In this version, I took it upon myself to create a deck of cards that a single player would have. The deck was a double pointer to an UnoCard. I used an array of pointers with each pointer holding the address of a single UnoCard object. That way, I could use a double pointer to represent a full deck for a single player. As far as functions are concerned, nothing was added.

Version 4: Deck Class

In this version, I created a Deck class that would hold a double pointer to an UnoCard as well as the number of cards a player has. This would take whatever I had in Version 4 and allow me to create a single Deck object that would hold a full deck of cards and the number of cards available. In order to do so, I had to use aggregation to make an UnoCard double pointer a member of my Deck class. I made a display function that would display not just a single card but the whole deck of a single player's cards.

Version 5: Card Matching

In this version, I got the basics of card matching down. If you recall, cards can be matched by colors, values, and descriptions. I made sure to process these properties, so that a match could be made in one of these scenarios. I added an isMatch() function in my Deck class that tested two

cards against each other with respect to their colors and values. I also created a copy card function that copied a card's contents over to another card. The purpose of this version was to get the card matching basics down and the interactions up and running.

Version 6: Card Matching

In this version, I made it so that we could match multiple times. The previous version simply matched a card once. I added 2 more functions that served as match checking functions. The isDrw1() tests two cards against each other to see if a draw 1 card can be matched. The isDrw2() does the same but for draw 2 cards. I then overloaded my first operator (+=). This operator was used to draw cards. If 1 card was drawn, I used deck+=1, to specify that we were adding 1 card to the deck. If we were drawing 2 cards, I used deck+=2, to specify that we were adding 2 cards to the deck. The purpose behind this version was to get a draw card function working and to throw it all into a loop where cards could be matched for as long as needed.

Version 7: Player Class

In this version, I created a Player Class which took everything and packaged it all together nicely. A Player object had a Deck Object as its member as well as a name. This Deck object was nothing more than an UnoCard double pointer if we think back. In other words, we have a player with a deck of UnoCard objects that can now be used as one entity. I created two Deck objects and set the cards up. I then used a new function that I wrote for the Player object to set the deck as our deck member. That way, we could now work with this data, display it, and modify it when needed. The process was as simple as creating and setting up a deck of cards and setting this deck as our deck member. I overloaded 2 more operators (-- and [ ]). The first would be used to decrement our number of cards when a card would be used, and the second would return an UnoCard based on the subscript we passed. For example, if we had deck[0], we'd be accessing the first card in our deck and the overloaded operator would return this data to us. We had to overload the [] operator because we used it with a deck pointer. We not only had to dereference the pointer from an array of pointers, we also had to dereference that individual pointer to work with an UnoCard object.

Version 8: Abstract Player to Player Class

In this version, I made an abstract player class which I used to derive my player class. Since the abstract class couldn't have any function definitions, it made it as easy as simply inheriting the abstract player class as my player class already had their definitions. This version was the first

case of inheritance used in my program. In the next version, I'd demonstrate inheritance once more, but for a more simple purpose. The purpose of this version was to get an abstract class working with my player class. Since I already had the function definitions, incorporating the abstract class was a simple process, which is how I like my programs.

Version 9: Player Class to Derived Class

In this version, I created another class that was to be derived from the player class. As the name suggests, the DerPlayer class was a class that was derived from the Player class and one that inherited the attributes and members of all previously inherited classes. We now had a chain of inheritance: AbsPlayer to Player and Player to DerPlayer. This DerPlayer class was a simple class that was created to demonstrate polymorphic behavior in the upcoming versions. The DerPlayer had an overridden get name function which appended "Mr/s." to the name when this function was called. As a result , we were to expect that when the get name function was called, the proper function would be called when we showcased polymorphic behavior.

Version 10: Polymorphic Behavior

In this version, we demonstrated polymorphic behavior. We created a DerPlayer object but used a Player pointer to reference it. In other words, we had a DerPlayer object that was being used as a Player object. When the object was to be used; however, we were to expect that the proper functions were being called. I also wanted to treat the DerPlayer object as a Player because it inherited all of the Player's members and functions, so it too was a Player. I used a Player double pointer to hold the addresses of 2 Player objects. The first pointer held a Player object, and the second held the address of a DerPlayer object. This was where I showcased polymorphic behavior. I had to recover all the memory that was used to create such a double pointer.

Version 11: Using Templates

In this version, I templatized everything that had to do with our Deck Class. In our Deck class, I used a general data type "T" to represent an UnoCard or any other Card object that could be derived from an UnoCard. Wherever I had UnoCard, I replaced it with "T". As a result of tweaking this class, I had to inherit the templated class whenever I used it in any other class. The only difference with how the program functioned was the fact that I now had to specify what type of card object was to be used. I left it as an UnoCard object, but if I wanted, I could derive another card object and use it in an UnoCard object's place. This helped me build my experience with working with not only template functions but also with template classes. As mentioned

previously, if I were returning a Deck object, I would have to return a Deck<T> object which would just represent a Deck object of a certain card type.

Version 12: Adding Requirements

In this version, I threw in some requirements that I failed to include in my previous versions. For the sake of not complicating the program, I took version 10 and worked from there with the addition of requirements. After all, all I did in Version 11 was add templates and nothing more. I created a copy constructor that could be used to copy a deck object into another deck object. This was an example of constructor overloading (Having two constructors with the same name that performed two different tasks). I also created a friend function for the << operator. If the << operator were to ever be used with a DerPlayer object, it would display the name and age of the DerPlayer. I forgot to mention that protected members were used in both the Player and DerPlayer class. These protected members were used from as early on as Version 10. This meant that the members were inherited publicly in our derived classes. I also threw in a static member variable and function. The static member held the number of Player objects that were created in the lifetime of our program. The static member function simply returned that value when called.

Version 13: Adding More Requirements

In this version, I made sure to include some other requirements, such as the STL and exceptions. As far as using exceptions were concerned, I made use of vectors contained in the <vector> file and a random shuffle algorithm contained in the <algorithm> header file. I also brought in the <new> header file which defined the bad_alloc exception type. That way, I could catch exceptions that were thrown in the case of bad memory allocation, ushc as if a negative number were used. The program would immediately terminate if this exception were caught and an error message would be displayed.

Version 14: Finishing Touches

This version was the big finish. I took everything that I had and made it more compact and straightforward. I created a simple menu function that would display a set of options to choose from. I then created a game function that took the majority of my code and made it a one-liner. I could call a game function and it would execute hundreds of lines in one go and all I had to do was throw the program that I had into a function. I added an exception class to make up for what I didn't fulfill through the bad_alloc exception. I could now catch memory allocation errors as well as when cards went above their specified range [1-15].

Project 2: Pseudo-Code

```
/*
 * File:   main.cpp
 * Author: Aamir Khan
 * Created on May 24, 2021, 12:15 PM
 * Purpose:  Project 2 - Uno Version 14
 */


//System Libraries
//I/O Library
//Cstd. Library
//Ctime Library
//Namespace std of system libraries


//User Libraries
//Include DerPlayer File (.h)

//Function Prototypes
//List my 3 prototypes.

//Main -> Execution Begins Here

        //Initialize the Random # Generator Seed
        //Declare Variables.
        //Display the Menu.
        //Using a Switch Statement, Either Play the Game or Quit.
                //Start the Timer and Begin our Game.
                //Once the Game Ends, Ask the User For a Second Game.
                //Once the Game Ends, Ask the User For a Third Game.
                //End the Timer and Display the Time it Took to Go Through 3 Games.
                //Display the Winner.
        //Quit the Program.


//Exit main - End of the Program
```

Sample Screen Shots:

```
switch(option){
    case '1':{
        unsigned int beg=time(0); //Beginning Time
        //First Game.
        game(nWins);
        //Second Game.
        char ch;
        cout<<endl<<"Player a Second Game (Y/N)? ";
        cin>>ch;
        if(ch=='Y'||ch=='y'){
            game(nWins);
        }
        //Third Game.
        cout<<endl<<"Player a Third Game (Y/N)? ";
        cin>>ch;
        if(ch=='Y'||ch=='y'){
            game(nWins);
        }
        unsigned int end=time(0);  //End Time
        //Display the # of times Player 1 won.
        cout<<endl<<"Games took "<<end-beg<<" seconds."<<endl;
        if(nWins==1) //I'm a stickler when it comes to grammar.
            cout<<"Player 1 won "<<nWins<<" time throughout this program.";
        else
            cout<<"Player 1 won "<<nWins<<" times throughout this program.";
        break;
    }
    default: cout<<"Quitting Program...";
}
```

This section of our code handles the 3 games by calling a game function 3 times. We utilize a switch construct to determine the User's choice (Play the Game or Quit the Program).

Manageable Main:

Aside from this code, there are several other lines, but code in main is as concise and clear cut as the functions make it. Code in main is just about 40 lines long as a result of our efforts.

Note:

*The nWins variable we pass in holds the number of wins Player 1 has (Passed by Reference).

```
#ifndef ABSPLAYER_H
#define ABSPLAYER_H

//Abstract Player Class
class AbsPlayer{
    public:
        //Pure Virtual Functions
        virtual void setDck(Deck *)=0;
        virtual void prntDck()=0;
        virtual string getName()=0;
        virtual int getnCards()=0;
};

#endif /* ABSPLAYER_H */
```

This is the abstract player class we use to derive our Player class and our Derived Player Class. Notice the =0 notation on our functions. This means that we must define these functions in our derived classes, but not here. As a result, we only have an AbsPlayer.h file and not a .cpp file.

```
#ifndef PLAYER_H
#define PLAYER_H

#include "Deck.h"
#include "AbsPlayer.h"

class Player:public AbsPlayer{
    protected:
        static int nPlyr; //The # of Players throughout our program.
        string name;      //Name of the Player
        Deck *deck;       //Their deck of cards.
    public:
        //Constructor
        Player(string);
        //Getter Functions
        string getName(){return name;}
        int getnCards();
        //Print Deck Function
        void prntDck();
        //Set Deck Function (Takes a Deck Object)
        void setDck(Deck *);
        //Static Member Functions
        static int getnPlyrs(){return nPlyr;}
};

#endif /* PLAYER_H */
```

This is our Player Class which is derived from our AbsPlayer Class. Notice the line in which we declare the class. This tells us that Player is derived from AbsPlayer.

```
//A Derived Player that's Used as a Player Object.
DerPlayer oppt(name2,age);
//An Array of Player Objects.
Player **plyrs=new Player *[nPlyrs];
//The first object is us.
plyrs[0]=new Player(name1);
//This showcases polymorphic behavior.
plyrs[1]=&oppt;
```

We create a Derived Player Object and reference it using a Player Pointer. This demonstrates polymorphic behavior because DerPlayer is derived from Player.

```
//Exception Class that is thrown if we have more than 15 cards.
class BadnCrds{};
//Constructor that takes the size of the deck.
Deck(int);
//Copy Constructor
Deck(Deck &);
//Destructor (Reallocates our Memory).
~Deck();
```

We've got not only constructors and destructors, but also a copy constructor as well as an exception class that is used to tell us when we have more cards than our range accepts.

```
//Overloaded [] Operator
UnoCard &operator[](int);
//Overloaded -- Operator
Deck operator--(int);
//Overloaded += Operator
Deck operator+=(int);
```

We overload three operators in our program. We first overload the [ ] operator to handle our Card objects. We then overload the -- operator to deal with discarding cards. We then overload the += operator which we use to draw cards whenever a draw 1 or 2 card is used.

```
//Ternary Operator
nCards=(n<=15?n:throw BadnCrds());
```

An exception class object being thrown.

```
//To determine what colors the cards are assigned.
vector<int> vect1;          //Create a Vector of Ints
for(int i=0;i<9;i++){       //Push the Values 1-9
    vect1.push_back(i+1); //Into our Vector.
}
//Randomize The Elements For Truly Random #'s
random_shuffle(vect1.begin(),vect1.end());
```

Using vectors from the STL and the random shuffle algorithm to set our Uno cards.

```
template <class T>
class Deck{
    private:
        int nCards;     //Stores the # of Cards a Player Has.
        T **deck; //The deck of cards that belong to the player.
    public:
        //Exception Class for Error Catching
        Deck(int); //Constructor that takes the size of the deck.
        ~Deck();   //Destructor (Reallocates our Memory).
        void setDck(); //Set Deck Function
        void dsply();  //Display Function for Deck Object
        //Getter Function.
        int &getnCards(){return nCards;}
        //Match Checking Functions
        bool isMatch(T,int);
        bool isDrw1(T,int);
        bool isDrw2(T,int);
        //Overloaded [] Operator
        T &operator[](int);
        //Overloaded -- Operator
        Deck<T> operator--(int);
        //Overloaded += Operator
        Deck<T> operator+=(int);
};
```

In Version 11 of our program, we create a Deck template class shown above. Wherever we would've used an UnoCard, we replaced it with the general data type "T". If we wanted to change it back, we could remove the T's and substitute UnoCard back in. This allows us to work with any objects that are UnoCards or that are derived from the UnoCard Class.

```
Welcome to the World of Uno...
Uno is a card game all about matching cards through colors, values,
and even playing a little dirty. Players have to compete to see who
can finish with their deck of cards first. Players can force
opponents to draw cards by pulling Draw 1 or 2 Cards. Furthermore,
Players can set new colors to match by pulling Wild Cards. A match
can be made by using cards with the same color, number, or description.
Descriptions include: Draw 1's, Draw 2's, and Wild Cards.

Choose an Option:
Option 1: Play Game.
Default: Quit Program.
1
Enter Player's Name: Aamir
Enter Opponent's Name: Mark
```

```
Card To Match...
Card 1 = [Blue  3]

Aamir's Cards To Choose From...
Card 1 = [Blue  4]
Card 2 = [Red  9]
Card 3 = [Yellow  9]
Card 4 = [Red Draw 2]
Card 5 = [Blue  3]
Choose a Card (Ex. 1 - Card 1):
```

Displaying our cards and asking the player for an input. This card that will be chosen can either match or fail to match. The card to match is shown at the top and our cards are displayed. We are then prompted to choose a card within our range of cards.

```
Card To Match...
Card 1 = [Red  7]

Aamir's Cards To Choose From...
Card 1 = [None Wild Card]
Card 2 = [Red Draw 2]
Card 3 = [Blue  1]
Choose a Card (Ex. 1 - Card 1): 2

Two Card Were Drawn By Mr/s.Mark...
```

The card to match is a Red 7. We choose our second card from our deck which happens to be a Red Draw 2. As a result, a match is made and Mr. Mark is forced to draw 2 cards. When his turn comes, he'll notice that there are 2 more cards in his deck due to this move.

```
Card was successfully matched!

Winner: Aamir
There were 2 Players in this Program. (Static Member)


Player a Second Game (Y/N)? n

Player a Third Game (Y/N)? n

Games took 12 seconds.
Player 1 won 1 time throughout this program.
RUN SUCCESSFUL (total time: 12s)
```

We played one game where I won. I display the static nPlyrs variable which holds how many Player objects were created in the lifetime of our program. I chose not to play the second and third game. The time it took for this all to take place is displayed.

```
Enter Player's Name: Aamir
Enter Opponent's Name: Mark
Enter the Opponent's age: 45
How Many Cards Do You Wish To Generate [1-15]? 20
ERROR: Too Many Cards. Card Range [1-15].
```

We are asked for the number of cards we wish to start with. The range is given, so when we exceed this range, an exception is thrown and caught.

We also have another exception that is thrown if we fail to allocate memory properly. The exception type is defined in the <new> header file and will display an error message if caught.

Project 2: Check-off Table

Chap.	Sect.	Topic	Line #"s	Pts	Notes
13		Classes			
	1 to 3	Instance of a Class	Line 107	4	Instance of a Deck Object.
	4	Private Data Members	Lines 15-16	4	Deck.h
	5	Specification vs. Implementation	Separate Files	4	.h and .cpp files for classes
	6	Inline	Lines 25-30	4	DerPlayer.h
	7, 8, 10	Constructors	Lines 21 & 23	4	Default Constructor & Copy Constructor in Deck.h
	9	Destructors	Line 25	4	Deck.h
	12	Arrays of Objects	Line 117	4	main.cpp (Part of Game Function)
	16	UML	Draw.io File And PDF	4	Found in Flowchart + Writeup Folder for Project 2
14		More about Classes			
	1	Static	Lines 16 & 30	5	Player.h
	2	Friends	Lines 14 & 33	2	DerPlayer.h
	4	Copy Constructors	Line 23	5	Deck.h
	5	Operator Overloading	Lines 35-40	8	Overloaded 3 operators ([ ] and -- and +=)

	7	**Aggregation**	**Line 18**	**6**	**Player.h**
15		**Inheritance**			
	1	**Protected members**	**Lines 16-18**	**6**	**Player.h**
	2 to 5	**Base Class to Derived**	**Line 12**	**6**	**AbsPlayer.h (Inherited by Player and DerPlayer)**
	6	**Polymorphic associations**	**Lines 115 & 121**	**6**	**DerPlayer object referenced by Player Object Pointer**
	7	**Abstract Classes**	**Line 12**	**6**	**AbsPlayer.h**
16		**Advanced Classes**			
	1	**Exceptions**	**Line 19**	**6**	**Exception Class (BadnCrds{})**
	2 to 4	**Templates**	**Lines 13-130**	**6**	**Deck.h Template Class Found in V.11 of my Project 2**
	5	**STL**	**Lines 26 & 31, 49 & 54**	**6**	**Vectors and Random_Shuffle Algorithm in STL**
		Sum		**100**	

Project 2: Flow-Chart & UML

(Found in FlowChart + Writeup Folder)